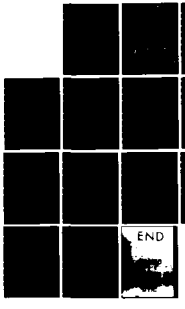
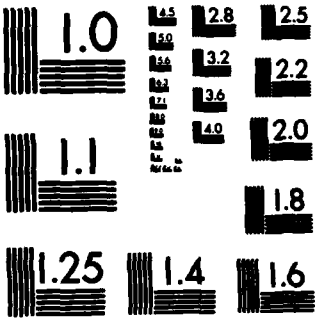
AD-A144 270 APPLICATION OF SOFTWARE TEST TOOLS TO BATTLEFIELD AUTOMATED SYSTEMS PHASE..(U) ARMY ARMAMENT RESEARCH AND DEVELOPMENT CENTER DOVER NJ PRODUC.. P E JANUSZ ET AL. 1/1

UNCLASSIFIED JUL 84 ARPAD-TR-84003 SBI-AD-E401 208 F/G 9/2 NL





(12)

AD

AD-E401 208

TECHNICAL REPORT ARPAD-TR-84003

AD-A144 270

APPLICATION OF SOFTWARE TEST TOOLS TO BATTLEFIELD AUTOMATED SYSTEMS, PHASE I

PAUL E. JANUSZ
WILLIAM R. TUROCZY

JULY 1984

U.S. ARMY ARMAMENT RESEARCH AND DEVELOPMENT CENTER
PRODUCT ASSURANCE DIRECTORATE
DOVER, NEW JERSEY

The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

The citation in this report of the names of commercial firms or commercially available products or services does not constitute official endorsement by or approval of the U.S. Government.

Destroy this report when no longer needed. Do not return to the originator.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Technical Report ARPAD-TR-84003	2. GOVT ACCESSION NO. ADA144270	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) APPLICATION OF SOFTWARE TEST TOOLS TO BATTLEFIELD AUTOMATED SYSTEMS, PHASE I		5. TYPE OF REPORT & PERIOD COVERED Final 16 Sep 82 - 31 Dec 83 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Paul E. Janusz William R. Turoczy		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS ARDC, PAD Technology & Automation, Information & Mathematics Div, [DRSMC-QAS-A(D)], Dover, NJ 07801-5001		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 612105H840011 Task No. VI-2
11. CONTROLLING OFFICE NAME AND ADDRESS ARDC, TSD STINFO Div [DRSMC-TSS(D)] Dover, NJ 07801-5001		12. REPORT DATE July 1984 13. NUMBER OF PAGES 42
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Director U.S. Army Materials & Mechanics Research Center ATTN: DRXMR-S Watertown, MA 02172		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES This project has been accomplished as part of the U.S. Army Producibility and Reliability for Readiness Program which has as its objectives the improvement of the life cycle reliability of Army systems through exploitation of emerging technologies. (cont)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Test driver Automated tools Embedded computers Testing methodologies Battlefield automated systems Software reliability Requirement verification & validation System safety Integrated logistics support		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report describes the first phase of a multi-phase research and development automated software test driver project. In recent years, much effort has been expended by researchers to develop practical software test tools to improve the reliability and reduce the prodigious life cycle costs of operational software. Much progress has been made within the last five years, and many of these tools have now emerged from the research stage to practical products. It is now feasible to acquire certain candidate test tools, evaluate their utility, and (cont)		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

18. SUPPLEMENTARY NOTES (cont)

Work generated by Dr. Martin L. Shooman was under contract DAAG29-81-D-10100.

20. ABSTRACT (cont)

demonstrate how they can be used to improve the software development process on a wide variety of Army projects.

The objective of phase 1 was to define and classify various categories of software test tools. Definitions and a glossary of terms are included for the reader's convenience. The basic features and principles of operation for test programs are introduced and discussed. It was concluded that these programs are practical and can be used as prototypes to introduce automated test tools into the Army software development process.

Based on the practicality of test tool usage and the potential for reliability improvement and cost savings, phase 2 will continue with the in-depth evaluation of the two most promising test-driver methodologies. These are the Cyclomatic Complexity Measure of McCabe & Associates and the Test Coverage Analysis Tool (TCAT) of Software Research Associates.

ACKNOWLEDGMENT

Much of the work reported here was generated with Dr. Martin L. Shooman of the Department of Electrical Engineering and Computer Science, Polytechnic Institute of New York. The authors would like to thank him for his guidance, assistance, and suggestions regarding the direction of this research work.

CONTENTS

INTRODUCTION

Many of the problems in developing reliable software for complex computer systems used in battlefield automated systems are related to defining software requirements. The high costs of software development and poor performance grow from ambiguous software requirements. The costs are directly related to the difficulties in defining software requirements, the problems with coordinating a large programming team, the difficulty of the test and error removal process, and the subsequent maintenance and enhancement of field deployed software.

At the National Conference on Software Development in Chicago which was sponsored by the Data Processing Management Association Educational Foundation in June 1983, Alfred Sorkowitz reported that the yearly cost of software developed in the United States was estimated to be over 20 billion dollars. (Other estimates from recent years ranged from 16 to 32 billion.) Furthermore, the Government Accounting Office studied nine federal software projects and determined that only approximately 2% of the software had been used as delivered to the government.

The Department of Defense has recognized the software testing problem, and document DoDD 5000.3 (Test and Evaluation) was established primarily to enforce the need for demonstrating software quality (refs 1-3). Special emphasis was placed on the testing and evaluating of software. Within DoDD 5000.3, the following four points are worth noting since they strongly relate to testing in the software development process:

1. Quantitative and demonstratable performance objectives shall be established for each software phase.

2. The decision to proceed to the next phase shall be based on quantitative demonstration of adequate software performance using test and evaluation.

3. Prior to release for operational use, software shall be operationally tested under realistic conditions to provide a valid estimate of system effectiveness and suitability in the operational environment.

4. Operational test and evaluation agencies shall participate in early stages of software planning and development to insure adequate consideration of the operational environment and objectives.

This study, directed at an examination of test tools, was encouraged by the recent progress on these tools (refs 4-6). The objective was to investigate the development of standardized methodologies for the design of software test drivers.[1]

---

[1] Test drivers are software programs which provide data for exercising and testing software that has been completed or is under development.

The long-term goal of this project is to develop standard methodologies and test drivers for the Army and perhaps other Department of Defense (DoD) agencies in order to assure that a uniform level of confidence for the software quality beyond the critical function stress tests is achieved. Stress testing is related to boundary value analysis which is a selection technique in which test data are chosen to lie along "boundaries" or extremes of input domain (or output range) classes, data structures, and other procedure parameters. Choices for data often include maximum, minimum, extraordinary, and degenerate values of parameters. As the long-term goal becomes achievable, there is a strong probability that a higher percentage of software will be used in the field with minor error corrections within the software development life cycle process.

The initial scope of work for this research project was prepared and submitted for approval in the last quarter of FY82. Briefly, the project funding levels were revised and an initial funding request was approved. In order to complete the first phase of the project, funds were approved for in-house government support and an additional amount was granted for outside consulting services.

The first phase was an investigation of the test driver literature and potential automated tools to explore, while the second phase is an experimentation with selected tools that will be evaluated for effectiveness in the software quality assurance functions for battlefield automated systems. This report presents the information discovered during the first phase of the research project. Proposals are being submitted for continued funding for the second phase.

Software Development Life Cycle

Software can be defined as the computer programs and data required to enable computer hardware to perform computational or data manipulation functions. Software is used in various environments including battlefield automated systems that are being investigated. Such software may include operational embedded software which is used for system performance, built-in-test, or test program sets which are used for the identification of malfunctions in the software and maintenance activities.

A simplified software life cycle for battlefield automated systems can be broken down into the following stages:

1. Conceptualization
2. Requirements defined
3. High level design
4. Detailed design
5. Implementation of design
6. Code testing
7. Requirements testing
8. Software maintenance

Throughout the system life cycle, there are various phases for testing, verifying, and validating the software. Test drivers can be used in these various cycles or phases of project development. For example, coding or requirements testing is an ideal phase for verifying and validating software. In addition, the software maintenance phase is critical in the software development life cycle. Test drivers are necessary in software quality assurance functions in order to reverify and revalidate software changes. They are used in regression testing and developing test beds for software qualification and acceptance.

Role of Testing

Testing is described by individuals in different ways. For example, Webster uses the terms "critical examination, observation, or evaluation" when he defines testing. Other researchers and authors described testing from different perspectives. Glenford Myers (ref 7) explains testing as the process of executing a program with the intent of finding errors. Michael Deutsch (ref 8) perceives testing as the controlled exercise of program code in order to expose errors, and Goodenough defines testing as a process inferring behavioral properties of a program based on the results of executing the program in a known environment with selected inputs.

More technially, Miller and Howden (ref 4) explain that a unit test of a single module consists of a collection of settings for the input space of the module and exactly one invocation of the module. A unit test may or may not include the effect of other modules which are invoked by the module undergoing testing.

The role of testing in a DoD environment is to verify and validate the demonstration of computer program requirements for system performance. Verification can be defined as the process of assuring the consistency and completeness of the current phase with the previous phases of the system life cycle development process. Validation is the assessment of the final software product through test, evaluation, and demonstration to measure how well the software conforms to the established requirements.

More specifically, the role of testing is very important in DoD projects and is used throughout the life cycle management of the software. Testing ultimately is used in formal qualification testing and acceptance. This later phase in the life cycle process is used as a prerequisite for final acceptance of the software technical data package which contractors submit to DoD agencies for approval.

The following classification schemes for test techniques are discussed: performance testing, code walkthroughs, classification of errors and test types, static and dynamic testing, set theory, input-output space, graph theory, and structured testing.

Statement of Problem

The present underlying problem in software development in DoD agencies is that much of the software developed by contractors and delivered to the government has a very low probability of being used as delivered. This problem can be broken down into many steps and subproblems related to the life cycle management process. Clearly defining software requirements early in the project is one subproblem. Others can be identified when the life cycle management process is studied more closely. The background and details of the DoD life cycle process are given in reference 9.

This research focuses on the testing aspects of software development and assumes that requirements are clearly stated and defined for the computer programs being developed. The approach taken was to define a technical objective related to software test driver development and reliability for readiness. The objective of this research was to develop standardized methodologies for the design and use of software test drivers which make it possible to assure a uniform level of confidence for the software quality beyond the critical function stress tests.

The following section presents research which classifies various test techniques and provides background for a basic understanding of the current literature available.

Purpose

A test driver is a simulated program which passes data to the module under test and receives data which the module has processed. Test drivers are frequently used to test software and demonstrate that the developed software conforms to system requirements. Investigation and development of a standardized methodology for the design of software test drivers is needed to assure a uniform level of confidence for software quality. This research project was initiated to explore existing methodologies and evaluate their strengths and weaknesses. The technical goal is, therefore, to discover an optimum system testing procedure whereby a standardized testing baseline can be established. If successful, the Army and the Department of Defense will be able to use this standard methodology in the development of reliable software for battlefield automated systems, combining academic expertise with practical structured testing methods.

The program plan consisted of the following key tasks (table 1) which were accomplished during phase 1:

1. Prepare plan for phase 1
2. Initiate prebidders conference for consultant selection
3. Review meetings with expert consultant
4. Conduct literature search on test drivers
5. Identify experts in software testing arena
6. Identify significant parameters
7. Review and finalize current models

8. Determine applicability of models
9. Determine limitations of models
10. Analyze models and prepare recommendations
11. Prepare technical report

Phase 1 identifies significant testing models that were evaluated to operationalize a standard approach in the development of test drivers to assure high quality software. Phase 2 will address the implementation of phase 1 and identify a transition team of researchers who will pursue further "hands-on" development work.

CLASSIFICATION OF TEST TECHNIQUES

Background

Over the last decade, many different test techniques have emerged. At present, there is no superior technique; however, most of the commercial and experimental test programs are based upon one or two approaches to testing. Therefore, even if these early commercial test products are imperfect, the approaches upon which they are built must be classified as the most advanced in a practical sense. It is realized that these approaches must have potential if developers have invested significant funds. Some practitioners have used these tools with success, and many others are experimenting with these programs.

Some recent progress has been made in characterizing and defining the various test approaches (refs 4 and 6).

Specification Languages

Since many of the errors which occur in a project are due to ambiguous or imprecise specification of the problem, various researchers have been working on languages which would make the specification process more precise. Most of these techniques combine a formal specification langage with some aspects of an interactive definition mode on a computer and are interspersed with English language, comments, and some analysis capability.

The analysis generally attempts to check for completeness, consistency, and ambiguity of the definitions, constraints, and statements in the specification language. Some of the best developed techniques are the A-7 specification technique (ref 10) and the problem statement language and analyzer (PSL/PSA) (ref 11). Most large software developers now use some type of pseudo-code based program design language (PDL).

Comprehensive Software Development Methodologies

Other researchers have approached the problem of errors which occur between the problem description and the coding phases. This is done by providing a comprehensive tool which allows the researcher to go from the basic concept of the

problem to a specification and then automatically generate code based upon the rigorously defined specification. Such techniques are best described by reference to particular examples of methodologies, e.g., the Higher Order Systems (HOS) methodology (refs 12, 13), and the software requirement engineering methodology (SREM) (ref 14).

Performance Testing

The term performance testing normally applies to the definition, modeling, and measurement of the computer or computer system. Typical measures that are used are millions of instructions per second (MIPS), millions of floating point operations per second (MFLOPS), and jobs processed per minute. Analysis of this last measure includes the load on a multiprocessing or time-sharing system, the task assignment, and queuing aspects of the operating system.

Code Walkthroughs

A code walkthrough is an informal design review. The participants are a lead programmer, the designer of the code, and possibly one or more programmers. The procedure involves the program designer's explaining his design, the underlying algorithms, and the code to the remainder of the group. Some combination of flow charts, pseudo code, or other design representation, along with a blackboard presentation or overhead transparencies are used to explain the code design. The main advantage of this technique is that several people interact, and the process minimizes the great tedium and boredom of code reading.

Classification of Errors and Test Types

From the literature reviewed, various categories of errors have been identified. These errors can be type classified into the following areas:

1. Logic
2. Documentation errors
    a. Code
    b. Specifications
    c. Unit development folders
3. Overload or overflow of range
4. Timing errors
5. Throughput

6. Recovery errors

7. Support software errors (test program sets)

8. Hardware errors

9. Specifications and requirements

10. Compilation errors

11. Interface errors (software)

12. Data or data base errors

13. Incorrect reporting

Errors can be identified by various methodologies and testing strategies can be developed. Each testing method has advantages and disadvantages for various types of errors discovered. Universal test methods and procedures may be very difficult to standardize. An effective approach to testing may involve a feedback process where various strategies of testing are performed, results analyzed and then modified based on the particular errors that are discovered. This process of interactive testing, analyzing, and retesting based on types of errors detected, may be the most cost effective procedure to investigate in test driver formulation.

Classes of testing are associated with various errors. Classification can be segmented into the following types:

1. Type 0 - All instructions in code executed at least once (check list)

2. Type 1 - All paths force-executed at least once (simulated 100% coverage)

3. Type 1.5 - All paths force-executed, some naturally executed

4. Type 2 - All paths naturally executed at least once (path coverage 100%)

5. Type 3 - All paths naturally executed for all values of input parameters (exhaustive test)

6. Type 4 - All paths naturally executed for all values of input parameters, all sequences of inputs, and all combinations of initial conditions (exhaustive test for multiprocessing, multiprogramming, and real time systems with nonfixed input sequence)

A more complete discussion of these classifications of testing is given in reference 15.

Static and Dynamic Testing

At the present time there are several testing methodologies, one of which is static testing. A static analysis is defined as a direct analysis of the form and structure of a product without executing the product (ref 16). This may include code inspection techniques, code walkthroughs, or symbolic analysis. Natural inputs are not applied to the program since this form of testing does not involve execution or run-time. This strategy in testing may be viewed as an extention of the compilation process (ref 4). Static testing can also be thought of as making certain allegations about the program and then proving these allegations. Once the discovered errors are corrected and static testing is completed, the next phase may include dynamic testing.

Dynamic testing involves execution or simulation of a development phase product. It detects errors by analyzing the response of a product to sets of input data (ref 16). In addition, dynamic testing is used for software qualification in the production phase of the life cycle. Dynamic testing is described in terms of the following four elements (ref 4):

1. Set an objective for the test which will usually show that the program is running according to the design requirements.

2. Set up appropriate input data for testing objectives.

3. Review output and results according to test objectives.

4. Develop measurements which quantify results and knowledge obtained about the program by executing the program.

By executing the program in a controlled environment, dynamic testing is used to demonstrate that errors are detected and eliminated. It assures that the program operates in the required fashion to detect unwanted or unnecessary functions.

Set Theory Analyses

Another testing methodology uses the concepts of set theory. There are generally two "solution" spaces, one for input and the other for output. The methodology involves mapping each element of the input space into a corresponding output space. The concept is that each input and output should be accounted for, and there should not be one without the other. If such an input or output is found to exist, it means that either a mistake in program logic exists (unnecessary data), or possibly an "added feature" is present. The solution spaces can also be broken down into corresponding domains, subsets, or subspaces.

This testing methodology is described in reference 17 in detail. Briefly, the subset or subspace is usually one of the program's paths. By using this methodology, certain classes of errors and programs can be defined. Errors include domain errors, computation errors, and subcase errors. Programs may be reliable, almost reliable, feasible, or infeasible. (For further details on the methodology, see reference 17.)

A methodology called functional program testing is described in reference 18 and is similar to the one just mentioned. According to this methodology, a program is viewed as a collection of functions each having input values for its variables and corresponding output values. The domains for the input and output are formally specified. The data contained within each of these domains corresponds to various functions and they may have one or more important properties. Test data are determined by considering these properties and reviewing the defined domains. (For additional depth on functional program testing, see reference 18.)

Graph Theory

Graph theory describes program logic by mapping data flow and control by means of flow charts and data flow diagrams. The flow chart is used to define program paths and derive test cases to test these paths. An automated method which develops an optimal set of test cases where all branches of the source code are exhaustively tested is described in reference 19. This process uses a minimum set of paths which cover all logical branches of the module. The testing process shows whether or not the program successfully performs its intended function in an acceptable fashion (e.g., efficiently, timely) for every combination of input variable values and every conceivable program path given.

The graph theory method consists of the following procedures: The source code is analyzed for its syntax. Certain branch conditions are identified as being incompatible. These incompatibilities may render a path unexecutable and they can be eliminated. However, some branch conditions may not be detected. This may require that the user input some information. The user then generates test data which will exercise all paths. A path which contains the maximum number of branches not yet exercised is identified. A test case to drive this path is generated in order to exercise the path. The process is repeated until all of the branches are exercised. Additional information on this methodology can be obtained from reference 19.

Structured Testing

The test methodology of structured testing relates to the concepts of graph theory. McCabe & Associates (ref 20) have done extensive work in this field. By determining the data flow diagram for a program under study, McCabe has developed a metric which gives an indication of the complexity of the program. The value of the complexity metric gives the minimum number of distinct paths (basis paths) which must be tested to assure software reliability. If the module or procedure has a cyclomatic complexity greater than ten, then the module is more likely to have errors. Furthermore, the module will be more difficult to understand, test, and debug. In addition, inherent timing problems may result, and the module will be difficult to maintain. To assure ease of understanding and testability, modules should be broken down into simpler components, each having a complexity of ten or less.

Modules or redesigned modules should be:

1. Testable in the sense that the testing effort can be managed properly.

2. Comprehensible so that the user can easily read and understand what is being done.

3. Definable so that it may be used in another system.

4. Maintainable to facilitate proper management.

EVALUATION OF TEST APPROACHES

Comparison of Features

The various test techniques which were introduced cannot be applied to all phases of software life cycle development with equal ease or validity. A comparison of the effectiveness of these various approaches and features appears in table 2.

The following main points from the table should be considered:

1. Many of the techniques involve a human analyst interacting with other key personnel during the life cycle management process or examining some stage of the development cycle. The elimination of the human factor element from the testing process cannot be envisioned. However, the most desired objective is to develop an automated tool or technique which will provide an additional means of finding errors and reducing the effort and the large number of manhours involved in testing a large program in battlefield automated systems.

2. The first priority would be to reduce further work on code reading, code walkthroughs, design reviews, program proofs, set theory analysis, and functional program testing since these areas are too complex and impractical for automation.

3. Much work is already in process on specification languages and comprehensive methodologies. At a later stage, these avenues will be explored further based upon state-of-the-art progress.

4. Since structured testing and test path approaches have been automated, they appear to be the most promising approaches for phase 2. Some of the features of symbolic testing have been incorporated in the programs which exercise test paths. Current trends in software development packages indicate that programming languages such as PASCAL and Ada are being used for current Department of Defense projects. These structured programming languages and techniques are forcing contractors to develop structured testing methodologies in order to qualify their software packages.

Based on these ideas, the following sections present the linkage between structured testing and automated tools. Structured testing is further elaborated upon with respect to the cyclomatic complexity metric developed by McCabe & Associates, Inc. The automated tool considered for further evaluation for phase 2 is the Test Coverage Analysis Tool (TCAT) developed by Software Research Associates.

Control Graphs

Some of the most promising software test tools developed to date are related to path testing of the software. The simplest way to define software paths is to refer to the control graph for the software. An approximate notion of the control graph can be quickly obtained if the flow chart for the software is considered. If each start-stop oval, processing box, input-output rhombus, and decision diamond are replaced by a small circle, then the nodes of the graph are defined. The flow lines in the flow chart that form the branches of the graph are retained. (In mathematical terms, the nodes and branches are called vertices and arcs.) In general there are two types of graphs, directed (digraphs) and nondirected. In a directed graph, the branches can only be traversed in the indicated arrow direction. In an undirected graph the branches can be traversed in both directions. If computer programs are modeled, directed graphs must be used.

Computer programs are initially considered without loops, therefore the flow charts do not have loops, nor do the associated control graph. If only structured programs are considered, then loopless programs contain only SEQUENCE and IF THEN ELSE control structures and do not contain DO WHILE control structures. Such a control graph contains only two-way brances at each IF THEN ELSE structure, which creates the paths. A path is a unique sequence of branches which connects the start and stop nodes in such a graph. A path test would generate test data to drive the program down all or some subset of all the graph paths.

If program loops (add DO WHILE or DO UNTIL) control structures in the code are allowed, the flow chart and the associated control graph will contain loops. Loops make the definition of paths more difficult, since every additional "trip around a loop" will create a new path. To avoid this problem, a loop in a program is defined as generating two paths: one is when the WHILE condition is initially false, and the loop is not executed; the other is when the loop is executed only the first time (ref 6).

Cyclomatic Complexity

A complexity measure for the control structure of a computer program by adapting graph theory was devised by McCabe (ref 20). This is known as the cyclomatic complexity of the graph and is related to the number of "independent" loops in the graph. The complexity can be calculated in several fashions, one of which is by adding unity to the difference between the number of branches and the number of nodes of the graph (refs 6 and 20).

McCabe further relates the cyclomatic complexity to the minimum number of test paths which are needed to cover (pass through) all the branches in the graph of the program. This technique can be implemented to generate the minimum set of tests for exercising the program and assuring software reliability (refs 6 and 20).

Test Coverage Analysis Tool

A test tool called Test Coverage Analysis Tool (TCAT) has been developed and marketed by Edward Miller of Software Research Associates.[2] This tool is based on path testing and generates a set of test conditions which drives the program down its various paths. Miller claims that although the tool will not generate test conditions for all the program paths, it will find and test up to 85% of the paths (test coverage of 0.85) in examples which he has investigated. It also provides a static analysis of the program along with the dynamic path testing.

This tool is available in several versions which run on a few different computers. TCAT appears to be one of the more advanced and flexible tools which are commercially available. Because of these strengths, it is recommended that one or more versions of TCAT be acquired for phase 2 of this research project, and that its capabilities and limitations be thoroughly explored.

Cyclomatic Complexity Measure in Testing

McCabe's cyclomatic complexity measure can be used to supplement a path testing program such as TCAT. The cyclomatic complexity can be computed and used as a lower bound on the number of path tests necessary for testing the program in question. It is advantageous to know if there are tens, hundreds, or thousands of test paths necessary before TCAT or similar test programs are used.

Another use of the cyclomatic complexity measure is in code inspection. A large number of programs and program modules can be submitted to a cyclomatic complexity analysis program and the various programs ranked with respect to their cyclomatic complexity. Rewriting or further study of the most complex programs in the group would then be considered.

Many private concerns are using automated tools to aid in their software development and analysis. For example, Ultrasystems from Irvine, California, has a number of automated tools in operation. One tool is a JOVIAL Code Analysis Program (JCAP). JCAP is a static analysis tool which provides information on standards auditing, structure analysis, data usage, flow analysis, and complexity metrics. Similar to JCAP, an Assembly Code Analysis Program (ACAP) is available to use for static analysis of assembly code.

---

[2] Correspondence between Miller and authors.

Another automated tool which is still in development by Ultrasystems is Multi-Language Static Analysis Tool (MSAT). The MSAT program structure analysis provides information on procedure structure call charts, intramodule charts, and nesting level of procedures. The data flow analysis examines module coupling analysis, global cross reference, constant usage, data usage, input/output reports, intramodule input/output reports, module strength analysis, and program stability analysis. The MSAT auditing tool investigates standards compliance. It provides complexity metrics and error analysis which do global data checks, constant checks, and provide reports on system completeness, procedure completeness, and procedure usage errors. The MSAT tool provides general system information on listings, procedure maps, assembly percentages, comment percentages, and abstracts. It is being developed to examine software changes. Code change analysis and structure change analysis will be provided in the MSAT tool. According to Ultrasystems, this tool is still under development but may be completed within a year.

Cyclomatic complexity analysis is also used in business application programs. AT&T in Warrenville, Illinois has expanded their software quality assurance group in order to provide feedback to designers of software programs on errors detected through cyclomatic complexity analysis and other automated test tools.

CHARACTERISTICS OF TOOL

After reviewing additional testing methodologies and techniques, general characteristics for a potential generic automated tool were formulated. Attributes to consider are:

1. User requirements and wants
2. Tool objectives
3. Tool usage
4. Tool languages
5. Primary features
6. Cost

As a pilot study, a simple problem was analyzed to determine characteristics of an automated tool. This analysis was performed manually, keeping in mind the process involved and how the computer/tool could be used to perform the same task. The steps followed in this procedure consisted of developing a flow chart, enumerating the number of paths, labeling the paths by some means to distinguish one from another, and generating data to drive a certain path.

User Interface

In developing a generic tool, the question of user interface comes to mind. The tool could be adaptable to a number of program languages. This would depend on the resources and/or hardware system capabilities available to the user. The coordination between user requirements and company policies needs to

be considered. To facilitate the use and management of the tool, a user's manual should be provided with typical examples including a complete record of the test procedures. Preferably, there should be a concise version of the user's manual on an interactive computer terminal which includes a user friendly HELP program menu.

User Friendly

One practical aspect of the tool would be to make it user friendly. User menus and prompts can guide the user through tool usage with a simplified set of instructions. Having the user interact with the tool would make it convenient and easy to understand what has been done or what will be done. It can also give some indication as what to do next. This feedback process allows software to be examined and changed easily. For example, if the program is too complex and the number of paths too large to test, the tool could ask the user to pick a particular path he might want to test. It might also ask the user if a retest were needed. This potential retest option would increase user confidence in tool reliability through the retest enhancement. The tool could have the ability to give feedback on critical paths that should be taken next and what input should be used to exercise the path. In addition, an automatic data generator could drive the desired path.

Output and Graphics

Probably one of the most important features of the tool would be output. It may discuss the number of paths in the program, the labeling of these paths, the percentage of paths tested, how many times each of the paths was tested, or the relative complexity of the program. The output should be in a format which is easy to interpret and comprehend. The use of graphics would simplify this process. A tree diagram or flow chart could be used to illustrate program logic or data flow. This could easily show the various paths of the program. By numbering or lettering the nodes and/or decision points, the various paths can be defined. When a path is executed, the output could consist of the numbers or letters to delineate the program path. If graphics are involved, the particular path could be highlighted by means of colors, dashed lines, or some other convenient method. A bar graph may be used to show the various paths and the relative frequency or percentage of times each path was tested and/or executed.

Software Research Associates has several software testing support tools. TCAT[3] illustrates the tool characteristics mentioned above. It is UNIX-based and available in COBOL, RATFOR, SRTRAN, and BASIC. The tool analyzes the program and defines modules and segments. The tool tests approximately 85% of the paths and also tests the inputs and outputs of the program. One of the outputs consists of

---

[3] Correspondence between Ed Miller and authors.

listing each module and identifying the segments, if any, which are not exercised. The TCAT Summary Test Coverage Report lists the following critical items:

1. Module name
2. Number of segments
3. Number of invocations
4. Number of segments exercised
5. Percentage covered

On the TCAT Histogram Report, the segment number and the number of executions of the segment are given and displayed in a histogram format. Many other leading companies have similar tools which are commercially available; however, researchers need to be very selective in purchasing automated tools.

CONCLUSIONS

The objective of phase 1 was to investigate the developing standardized methodologies for the design of software test drivers. Test drivers are necessary to assure a uniform level of confidence for the software quality beyond the critical function stress tests. These standardized test drivers are being considered for use in battlefield automated systems within the Department of Defense.

A literature survey and personal interviews with leading software developers, such as AT&T and Ultrasystems, were conducted. Static and dynamic testing methodologies were investigated with respect to the advantages and disadvantages. As a result, a standardized automated tool for each application seems highly unlikely. Therefore, a generic strategy for developing a standardized test driver needs further investigation.

RECOMMENDATIONS

It is recommended that the most promising test driver methodologies be examined in phase 2. These include the Cyclomatic Complexity Measure of McCabe & Associates, Inc. and the Test Coverage Analysis Tool (TCAT) of Software Research Associates.

The long term objective is to develop a standard methodology that can be automated as a generic tool (test driver) to be used in Army and Department of Defense Software Quality Assurance Centers to increase software reliability. Strategies for tailored automated test drivers will be generated to assure increased confidence in software development throughout the life cycle of the battlefield automated system. To achieve this objective, continued research is proposed to supplement this effort. The planning for the next phase of the research is broken down into the following steps:

1. Investigate tools that are available, purchase the tools, and verify that the tools actually perform their intended function. In essence, check the

tools for content validity. As a result of this research, several tools can be validated and used. As previously mentioned, Software Research Associates have several automated tools. Other industries which have similar tools include Boeing, McDonnell Douglas, IBM, Bell Laboratories, and Softool. Within the government, the Federal Software Testing Center in Washington, DC has additional software tools available; however, not all of these tools are commercially available.

2. Study the fundamentals of the tool. This would consist of determining how the logic diagram, flow chart, or paths are defined. Basic factors such as what inputs are necessary to exercise a particular path and how to interpret the output are important. The result is a basic understanding of how the tool operates and experiencing its limitations.

3. Introduce a representative set of errors into the program. Since it is difficult to seed errors, an obsolete program with known errors can be used to check the validity of the tool. This effort would build on steps one and two. Knowledge and insight gained would help users to understand the fundamentals of the tool. Additional significant findings are then documented in the updated user's manual.

4. Summarize the research by modifying and consolidating the lessons learned from selected tools and create a prototype test driver.

5. Revise the user's manual for the tool. It is assumed that the tools are written by experts for use by experts. A user's manual would be developed in order that the user could fully understand the basic ideas, applications, and practicality of the tool in Army and DoD environments.

REFERENCES

1. MIL-STD-1679, Weapon System Software Development, December 1, 1978.

2. DARCOM-R 70-16, Management of Computer Resources in Battlefield Automated Systems, July 16, 1979.

3. AMC-R 700-38, Test and Evaluation Incidence Disclosed During Material Testing.

4. Edward Miller and William E. Howden, Tutorial: Software Testing and Validation Techniques, second edition, IEEE Computer Society, cat. no. EHO 180-0, IEEE, New York, 1981, 499 p.

5. J. T. McCabe, "A Complexity Measure," IEEE Trans. Software Engineering, vol. SE-2, no. 4, December 1976.

6. Martin L. Shooman, Software Engineering: Design, Reliability, and Management, McGraw Hill, New York, N.Y., 1983.

7. Glenford J. Myers, Software Reliability Principles and Practice, 1976.

8. Michael S. Deutch, Software Verification and Validation--Realistic Project Approaches, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

9. William R. Turoczy, "Software Quality Assurance (SQA) Life Cycle Management Course Notes and Info," ARDC, Dover, New Jersey, January 1984.

10. Robert A. Parker, et al., "Abstract Interface Specification for the A-7E Device Interface Module," NRL Memo. Report 4385, November 20, 1980.

11. Daniel Teichroew, "ISDOS and Recent Extensions," Proc. Symp. Comp. Software Eng., Polytechnic Press, Brooklyn, New York, 1976, pp 75-81.

12. H. Hamilton and S. Zeldin, "The Relationship Between Design and Verification," J. Sys. and Software, vol 1, Elsevier North-Holland, Inc., New York, 1979, pp 29-56.

13. Higher Order Software, Inc., The FAME Configuration Reference Manual, HOS, Cambridge, Massachusetts, December 1981.

14. Thomas Bull, David Bixler, and Margret Dyer, "An Extendable Approach to Computer Aided Software Requirements Engineering," IEEE Trans. on Software Engineering, vol SE-3, no. 1, January 1977, pp 49-59.

15. D. L. Baggi and Martin L. Shooman, "Software Test Models and Implementation of Associated Test Drivers," Report SRS-116/Polytechnic Institute of New York, November 1979.

16. U.S. Department of Commerce, National Bureau of Standards, "Software Validation, Verification, and Testing Technique and Tool Reference Guide," NBS Special Publication 500-93, September 1982.

17. William E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Transactions on Software Engineering, vol SE-2, no. 3, September 1976.

18. William E. Howden, "Functional Program Testing," IEEE Transactions on Software Engineering, vol SE-6, no. 2, March 1979.

19. K. W. Krause, R. W. Smith, and M. A. Goodwin, "Optimal Software Test Planning Through Automated Network Analysis."

20. U.S. Department of Commerce, National Bureau of Standards, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," NBS Special Publication 500-99, December 1982.

BIBLIOGRAPHY

1. D. L. Baggi and Martin L. Shooman, "An Automatic Driver for Pseudo-exhaustive Software Testing," Digest of Papers COMPCON '78, IEEE, New York, February 28, 1978, p 278.

2. Boris Beiser, Software Testing Techniques, Van Nostrand and Reinhold Co., New York, 1983.

3. L. E. Bonanni and A. L. Glasser, "Source Code Control System Programming Work Bench Unix (SCCS/PWB/UNIX) Manual," Bell Laboratories, Holmdel, New Jersey.

4. Fredrick P. Brooks, Jr., The Mythical Man-Month, Addison-Wesley, Reading, Massachusetts, 1975.

5. Robert C. Bruce, Software Debugging for MicroComputers, Reston Publishing Co., Reston, Virginia, 1980.

6. Data and Analysis Center for Software, "A Bibliography of Software Engineering Terms," Rome Air Development Center, RADC/ISISI, Rome, New York, October 1979.

7. Rajat K. Deb, "On Generation of Test Data and Minimal Cover of Directed Graphs," Proc. IFIP Cong. 1977, August 8-12, 1977, pp 13-16.

8. Tom Demarco, Structured Analysis and System Specification, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

9. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmers Workbench," Proc. 2d Int. Conf. Software Eng., IEEE, New York, October 1976, pp 164-199.

10. T. A. Dolotta, et al, "The LEAP Load and Test Driver," Proc. 2d Int. Conf. Software Eng., IEEE, New York, October 1976.

11. O. E. Ellingson, "Computer Program and Change Control," Rec. 1973 IEEE Symp. Comp. Software Reliability, IEEE, New York, pp 80-89.

12. W. R. Elmendorf, "Cause-Effect Graphs in Functional Testing," Report TR-00.2487, IBM Systems Development Division, Poughkeepsie, New York, 1973.

13. Doug Ferguson and Leon F. Young, "Structured Analysis and Testing of Divad Embedded Software."

14. D. R. Giloty, et al., "System Testing and Early Field Experience," Bell Sys. Tech. J., vol 49, December 1970.

15. Robert L. Glass, Software Reliability Guidebook, 1979.

16. Robert L. Glass and Ronald A. Noiseux, Software Maintenance Guidebook, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

17. Alan L. Glasser, "The Evolution of a Source Code Control System," Proc. Software Quality and Assurance Workshop, ACM, New York, November 1978, pp 122-125.

18. M. D. Godfrey, et al., Machine Independent Organic Software Tools (MINT), Academic Press, New York, 1980.

19. K. A. Heller, et al., "System Testing," Bell Sys. Tech. J., vol 49, no. 10, December 1970, p 2711.

20. William C. Hetzel, ed., Program Test Methods, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

21. Higher Order Software, Inc., The FAME Configuration Reference Manual, HOS, Cambridge, Massachusetts, December 1981.

22. IEEE Computer Society, Proceedings Soft Fair - A Conference on Software Development Tools, Techniques, and Alternatives, IEEE Computer Society, cat. no. 83CH1919-0, IEEE, New York, 1983.

23. IEEE Computer Society Committee on Software Engineering, Software Engineering Terminology, 1979.

24. Randell W. Jensen and Charles C. Tonies, Software Engineering, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

25. Brian W. Kernighan and P. J. Plauger, Software Tools, 1976.

26. Brian W. Kernighan and P. J. Plauger, Software Tools in Pascal, Addison-Wesley, Reading, Massachusetts, 1981.

27. Arthur Laemmel, "Dillworth's Theorem and Program Testing," unpublished memorandum, Polytechnic Institute of New York, Dec. 9, 1975.

28. Arthur Laemmel, "Notes on Digraphs and Programming," unpublished memorandum, Polytechnic Institute of New York, August 1975.

29. Arthur Laemmel, "Testing Flow Charts with Loops," unpublished memorandum, Polytechnic Institute of New York, December 10, 1975.

30. Arthur Laemmel, "A Statistical Theory of Computer Program Testing," Report SRS119/POLY EE 80-004, Polytechnic Institute of New York, June 1980.

31. R. Linger, H. Mills, and B. Witt, Structured Programming Theory and Practice, 1979.

32. Thomas J. McCabe, Structured Testing, IEEE Computer Society, cat. no. EH0200-6, IEEE, New York, 1983, 132 p.

33. Thomas J. McCabe and G. Gordon Schulmeyer, "System Testing Aided by Structured Analysis (A Practical Experience)," IEEE COMPSAC Proceedings, 1982, November 10-12, Chicago, Illinois.

34. S. N. Mohanty, "Automatic Program Testing," Ph.D. thesis, Polytechnic Institute of New York, Department of Electrical Engineering and Computer Science, June 1976.

35. John D. Musa, "Validity of Execution-Time Theory of Software Reliability," IEEE Transactions on Reliability, vol R-28, no. 3, August 1979.

36. Glenford J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity," SIGPLAN Notices, October 1977.

37. Glenford J. Myers, The Art of Software Testing, Wiley, New York, 1979.

38. G. S. Popkin and Martin L. Shooman, "On the Number of Tests Necessary to Verify a Computer Program," Report SRS-113/Poly EE 78-047, Polytechnic Institute of New York, June 1978.

39. Proceedings of the 1975 International Conference on Reliable Software, IEEE, April, 1975.

40. Proceedings of the Symposium on Computer Software Engineering, Polytechnic Press, New York, 1976.

41. Proceedings of the Software Quality Assurance Workshop, ACM SIGMETRICS, November 1978.

42. Proceedings of the First (-Sixth) International Conference(s) on Software Engineering, IEEE. Conference dates: September 1975, October 1976, May 1978, September 1979, September 1981, September 1982.

43. Wendy Rauch-Hinden, "Software Tools: New Ways to Chip Software Into Shape," Data Communications, April 1982, pp 83-113.

44. Record 1973 IEEE Symposium on Computer Software Reliability, April 30, 1973.

45. Donald J. Reifer and Stephen Tratter, "A Glossary of Software Tools and Techniques," Computer, July 1977.

46. J. Rubey, "Quantitative Aspects of Software Validation," Int. Conf. Reliable Software, IEEE, New York, 1975, p 246.

47. Randall Rustin, Debugging Techniques in Large Systems, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1971.

48. A. L. Scherr, "Developing and Testing a Large Programming System, OS/360 Time Sharing Option," in Hetzel (1973).

49. Martin L. Shooman, "Analytic Generation of Test Data," unpublished memorandum, Polytechnic Institute of New York, December 1974.

50. Martin L. Shooman, "Analytic Models for Software Testing," unpublished memorandum, Polytechnic Institute of New York, December 17, 1974.

51. Martin L. Shooman, "Meaning of Exhaustive Testing," Research Report EE/EP 74-006/EER 105, January 2, 1974.

52. Martin L. Shooman and Morris Bolsky, "Types, Distribution, and Test and Correction Times for Programming Errors," Proc. 1975 Int. Conf. Reliable Software, IEEE, New York, cat. no. 75 CHO 940-7CSR, p 347.

53. Robert C. Tausworthe, Standardized Development of Computer Software, Prentice Hall, Englewood Cliffs, New Jersey, Part I, 1977, Part II, 1979.

54. Thomas A. Thayer, et al., Software Reliability, 1978.

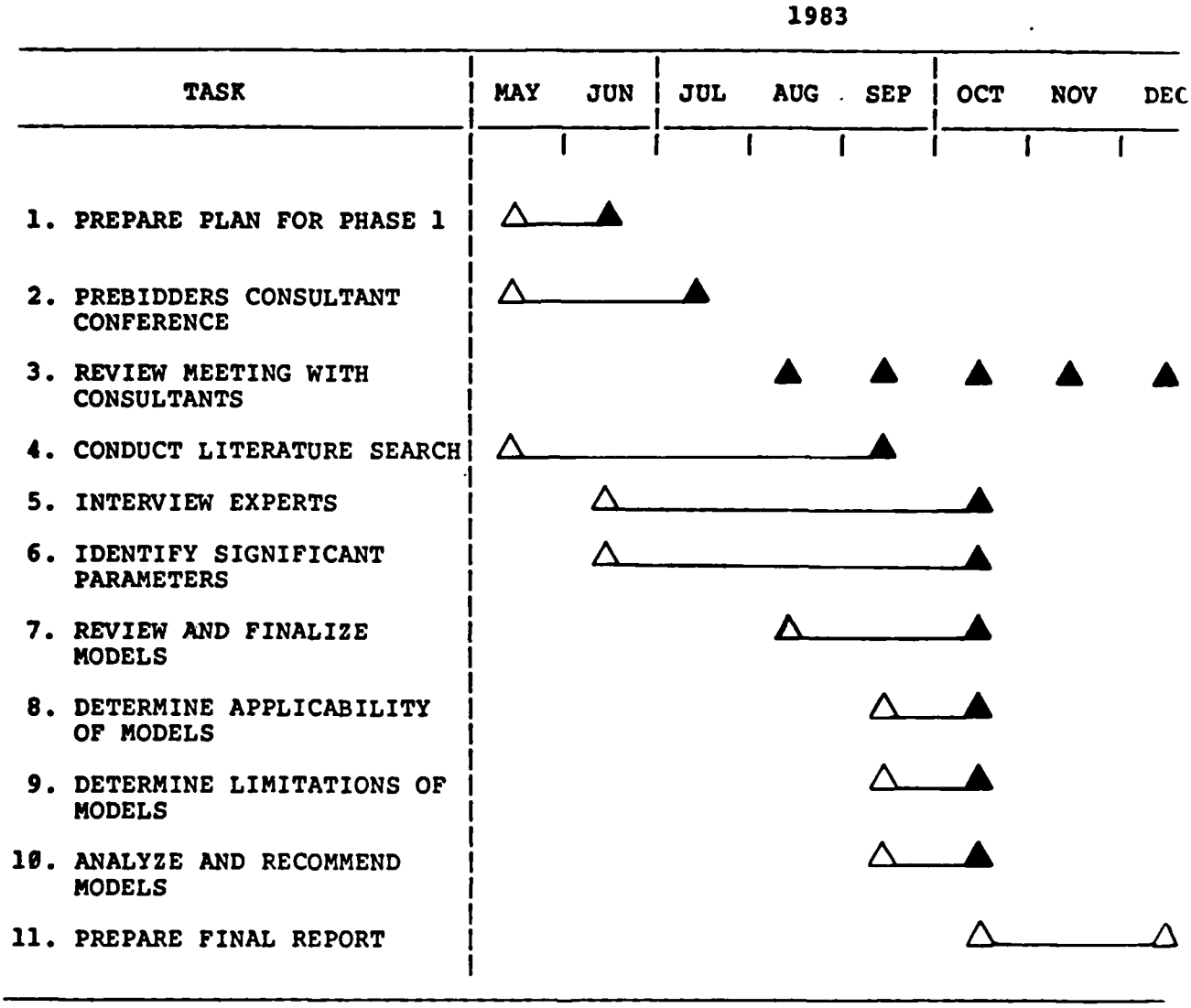
55. U.S. Department of Commerce, National Bureau of Standards, "The Introduction of Software Tools," NBS Special Publication 500-91, September 1982.

56. U.S. Department of Commerce, National Bureau of Standards, "Planning for Software Validation, Verification, and Testing," NBS Special Publication 500-98, November, 1982.

57. Martin R. Woodward, et al., "A Measure of Control Flow Complexity in Program Test," IEEE Trans. Software Eng., vol SE-5, no. 1, January 1977, p 981.

58. Marvin V. Zelkowitz, "Automatic Program Analysis and Evaluation," Proc. 2d Int. Conf. Software Eng., IEEE, New York, October 1976.

Table 1. Test driver R&D program schedule and milestones



LEGEND:
△ START
▲ COMPLETE
◊ PLANNED SLIP

Table 2. Comparison of various test approaches

Testing method	Advantages	Disadvantages	Comments
Static testing			Errors which depend upon timing or execution sequence are extremely difficult to find
Code reading	Inexpensive (manhours, CPU sec.)	Boring, laborious	Used as an adjunct with other techniques
Code walkthroughs	Same as above plus added insight due to group interaction	Tedious but not as bad as code reading	Good working group of lead programmer and one or more programmers of his group is necessary
Design review	Same as above, more formal, a wide range of reviewers and backgrounds	Costly (travel, manhours)	Effective in finding incompatabilities and specifications
Specification language	Requires the specification writer and contractor to use the same specification language	Requires the learning of another higher order language	Effective for improving the interface between the specification writer and the contractor
Comprehensive methodology	Similar to above but with added features of comprehensive methodology and code generation	Same as above but with additional investment of time and money	Relatively new concept in the experimental and development stages
Program proofs	Provides discipline and has a very low error rate	Requires high mathematical skill level, proof may be longer than program, cumbersome.	If technique becomes practical, it could be used to develop very low error rate software
Symbolic testing	Simulated execution of program path, provides an analytical trace of program	Requires some interpretation of results	Some of the same features as program proofs
Dynamic testing			Focuses on execution of program for known test cases and comparison of output with known results
Set theory analysis	Provides a functional relationship between inputs and outputs	Not always easy to define such relationships in all problem classes	Many programmers intuitively use some of these concepts to a certain extent in practice
Functional program testing	Incorporates concepts mentioned in input/output space and program proofs	High mathematical skill level required, detailed knowledge of program required	Considerable promise if it becomes practical
Test paths	Relates to graph model and provides a visual model of program execution	Detailed knowledge of program structure required	Many currently available test tools use this approach
Structured testing	Provides a quantitative measurement of program complexity	Very few comparison studies have been documented	Simple method for identifying program complexity; however, data base needs expansion to validate and explore the complexity measure

GLOSSARY

ACAP: Assembly Code Analysis Program.

ACCEPTANCE TESTING: The validation of the system or program to the user requirements.

BACKTRACKING: Examining the error symptoms to see where they were first noticed and then backstepping in the program flow of control to a point where the symptoms have disappeared.

BIG BANG TESTING: When all modules have been individually coded and are submitted to integration testing without prior unit testing.

BUILD: A version of a program incorporating a subset of the required features. Software goes through many builds as it evolves in a project.

CERTIFICATION: An authorative endorsement of the program. Testing for certification must be done against some predefined standard.

CLASSIC SOFTWARE DEVELOPMENT: Development which has proceeded bottom-up in design, coding, and testing.

CODE READING: The reading of a program, by another programmer other than the designer, with the purpose of finding errors.

CONFIGURATION CONTROL: The strict control of the source code of a program under development. The controller, called the configuration manager, keeps the official copy of the program. Changes or corrections can only be made on written request to the configuration manager (and the configuration control board).

CONFIGURATION MANAGEMENT: See Configuration Control.

COVER: A set of tests which cover (exercise) all instructions in a program. Might also mean a set of tests which exercise all control paths or flows in a program.

CYCLOMATIC NUMBER: For a control graph, the cyclomatic complexity number can be calculated by taking the number of edges (branches), subtract the number of vertices (nodes), and add unity.

DEBUGGING: The activity of diagnosing the precise nature of a known error and then the correcting of the error.

DEDUCTIVE DEBUGGING: Process begins by enumerating all causes or hypotheses which seem possible, then one by one, particular causes are ruled out until a single one remains for validation.

DESIGN REVIEWS: Formal meetings held by the project director for his representative, other professionals, and at times the customer's representative, to review in detail the progress of the project.

DESK CHECKING: See Code Reading.

DoD: Department of Defense.

ENHANCEMENT: The redesign of a significant portion of the software to provide additional, improved, or changed functions. Sometimes enhancement is wrongly classified as maintenance.

EXTERNAL FUNCTION TESTING: The verification of the external system as stated in the external specifications.

EYEBALLING: See Code Reading.

FIELD TESTING: The initial operation of the actual hardware or software system in the field in a test mode (limited or full capabilities) to ferret out as many errors as is feasible.

HAND EXECUTION: See Code Reading.

HOS: Higher order systems.

INDUCTIVE DEBUGGING: Approach comes from the formulation of a single working hypothesis based on the data, on the analysis of existing data, and on specially collected data to prove or disprove the working hypothesis.

INSTALLATION TESTING: The validation of each particular installation of the system with the intent of pointing out errors made while installing the system.

INTEGRATION TESTING: The verification of the interfaces among system parts (modules, components, and subsystems). Tests are performed which exercise interfaces among program modules.

JCAP: JOVIAL Code Analysis Program.

MAINTENANCE: The support of operational software through documentation of errors discovered in the field, generation of work around procedures, correction of errors (where appropriate), and installation of very minor changes and additions to the code (see Enhancement).

MFLOPS: Millions of floating point operations per second.

MILESTONE: A significant event in a PERT project management chart.

MIPS: Millions of instructions per second.

MODERN SOFTWARE DEVELOPMENT: Development which has proceeded topdown in design, coding and testing.

MODIFIED TOP-DOWN TESTING: While integration testing of the control structure is in progress the critical module is being exercised with a test driver program.

MODULE TESTING or UNIT TESTING: The verification of a single program module, usually in an isolated environment (i.e., isolated from all other modules).

MSAT: Multi-Language Static Analysis Tool.

PATH: A sequence of edges which when traversed in the arrow direction form a connection from the start vertex to the stop vertex.

PATH TEST: The traversal of a path.

PDL: Program design language.

PERT: Program Evaluation and Review Technique. A diagram (flow graph like) which includes milestones, events, and paths between them. A PERT diagram is used to document and manage program progress.

PROGRAM PATH: A graph that includes only a single traversal of any loops which are encountered.

PROOF: An attempt to find errors in a program without regard to the program's environment.

PSEUDO-CODE: A program design representation technique composed of a mixture of English language statements, generic programming statements, and a few statements specific to the chosen programming language. The purpose is to produce a sort of "pidgen English" which presses the main features of the design at a high level.

REGRESSION TESTING: Repetition, in whole or in part, of testing after an error has been discovered and a correction has been made.

SANDWICH TESTING: Top-down and bottom-up approach to testing, working from both ends toward the middle. Used if there is more than one critical module or if the critical module is totally persuasive.

SIMULATION TESTING: The exercise of the software in conjunction with a simulation program and often some peripheral hardware which replicate the real operating environment as closely as possible. The computer used to run the software and the simulation program may be the computer to be used in the field or a development computer.

SOFTWARE RELIABILITY: The probability that a program will perform it's intended task for a certain period without causing a system failure.

SQA: Software Quality Assurance.

SREM: Software Requirement Engineering Methodology.

STRONGLY CONNECTED GRAPH: A graph in which each node in the graph can be reached from any other node.

SYMBOLIC TESTING: An analysis technique which derives a symbolic expression for every path of the program.

SYSTEM INTEGRATION: The process by which individual modules are put together to realize major subsections and functions of a program.

SYSTEM TESTING: The verification and/or validation of the system to its initial objectives. System testing is a verification process when it is done in a simulated environment; it is a validation process when it is performed in a live environment.

TCAT: Test Coverage Analysis Tool.

TEST DRIVERS: A simulated program which passes data to the module under test and receives data which the module has processed. Input data or code used to check if the software meets the requirements and does its intended function. Needed in a bottom-up coded program.

TESTING: The process of executing (or evaluating) a program (or part of a program) with the intention or goal of finding errors. The activity of finding errors.

TEST STUBS: Output statements for each module within the control structure to indicate that control has passed through the module during the tests of the control structure. Needed in a top-down coded program.

VALIDATION: An attempt to find errors by executing a program in a given real environment.

VERIFICATION: An attempt to find errors by executing a program in a test or simulated environment.

WALKTHROUGH: An informal design review held by a lead programmer and some of his staff to explore any design or coding errors which may exist.

DISTRIBUTION LIST

Commander
Armament Research and Development Center
U.S. Army Armament, Munitions and
Chemical Command
ATTN: DRSMC-TSS(D) (5)
DRSMC-GCL(D)
DRSMC-LC(D), J. Lepore
DRSMC-LCA(D), A. Moss
DRSMC-LCE(D), R. Walker
DRSMC-QAF-I(D) (10)
DRSMC-QAS(D), B. Aronowitz (5)
DRSMC-QAS-A(D), Paul E. Janusz (10)
DRSMC-QAS-A(D), William R. Turoczy (10)
DRSMC-SCM(D), J. D. Corrie
DRSMC-SCM-O(D), H. Pebly, Jr.
DRSMC-TSP(D), B. Stephans
Dover, NJ 07801-5001

Administrator
Defense Technical Information Center
ATTN: Accessions Division (12)
Cameron Station
Alexandria, VA 22314

Director
U.S. Army Materiel Systems Analysis
Activity
ATTN: DRXSY-MP
Aberdeen Proving Ground, MD 21005

Commander/Director
Chemical Research and Development Center
U.S. Army Armament, Munitions and
Chemical Command
ATTN: DRSMC-CLJ-L(A)
DRSMC-CLB-PA(A)
DRSMC-OAC-E(A), W. J. Maurits
APG, Edgewood Area, MD 21010

Director
Ballistics Research Laboratory
ATTN: DRXBR-OD-ST
Aberdeen Proving Ground, MD 21005

Chief
Benet Weapons Laboratory, LCWSL
Armament Research and Development Center
U.S. Army Armament, Munitions and
Chemical Command
ATTN: DRSMC-LCB-TL
DRSMC-LCB, T. Moraczewski
SARWV-PPI, L. Jette
Watervliet, NY 12189

Commander
U.S. Army Armament, Munitions and
Chemical Command
ATTN: DRSMC-LEP-L(R)
DRSMC-EN(R)
DRSMC-QA(R) (2)
DRSMC-QAE(R)
DRSMC-RDP(R)
DRSMC-SC(R)
Rock Island, IL 61299

Director
U.S. Army TRADOC Systems
Analysis Activity
ATTN: ATAA-SL
White Sands Missile Range, NM 88002

Director
U.S. Army Materials and Mechanics Research Center
ATTN: DRXMR-D
DRXMR-P
DRXMR-PL (2)
DRXMR-S
DRXMR-STQ (4)
Watertown, MA 02172

Commander
U.S. Army Foreign Science and Technology
Center
ATTN: DRXST-SD3
220 Seventh Street, N.E.
Charlottesville, VA 22901

Office of the Deputy Chief of Staff for
Research, Development, and Acquisition
ATTN: DAMA-ARZ-E
DAMA-CSS
Washington, DC 30210

Commander
Army Research Office
ATTN: George Mayer
J. J. Murray
P.O. Box 1211
Research Triangle Park, NC 27709

Commander
U.S. Army Materiel Development
and Readiness Command
ATTN: DRCDE-D
DRCDMD-FT
DRCLDC
DRCMM-M
DRCMT
DRCQA-E
DRCQA-P
5001 Eisenhower Avenue
Alexandria, VA 22333

Commander
U.S. Army Electronics Research and
Development Command
ATTN: DRSEL-PA-E, Stan Alster
J. Quinn
Fort Monmouth, NJ 07703

Commander
U.S. Army Missile Command
ATTN: DRSMI-EAT, R. Talley
DRSMI-ET, Robert O. Black
DRSMI-M
DRSMI-QP
DRSMI-QS, George L. Stewart, Jr.
DRSMI-TB (2)
DRSMI-TK, J. Alley
Redstone Arsenal, AL 35809

Commander
U.S. Army Troop Support and Aviation
Materiel Readiness Command
ATTN: DRSTS-M
DRSTS-PLE, J. Corwin (2)
DRSTS-Q
4300 Goodfellow Boulevard
St. Louis, MO 63120

Commander
U.S. Army Natick Research and
Development Command
ATTN: DRDNA-EM
Natick, MA 01760

Commander
U.S. Army Mobility Equipment Research
and Development Command
ATTN: DRDME-D
DRDME-E
DRDME-G
DRDME-H
DRDME-M
DRDME-N
DRDME-T
DRDME-TQ
DRDME-V
DRDME-ZE
Fort Belvoir, VA 22060

Commander
U.S. Army Tank-Automotive Materiel
Readiness Command
ATTN: DRSTA-Q (2)
Warren, MI 48090

Commander
Rock Island Arsenal
ATTN: SARRI-EN, W. M. Kisner
SARRI-ENM, W. D. McHenry
SARRI-QA
Rock Island, IL 61299

Commander
U.S. Army Aviation Research and
Development Command
ATTN: DRDAV-EXT
DRDAV-QE
DRDAV-QP
DRDAV-QR
St. Louis, MO 63120

Commander
U.S. Army Tank-Automotive Research
and Development Command
ATTN: DRDTA-JA, C. Kedzior
DRDTA-RCKM, S. Goodman
DRDTA-RCKT, J. Fix
DRDTA-RTAS, S. Catalano
DRDTA-TTM, W. Moncrief
DRDTA-UL, Technical Library
DRDTA-ZE, O. Renius
Warren, MI 48090

Commander
Harry Diamond Laboratories
ATTN: DELHD-EDE, B. F. Willis
2800 Powder Mill Road
Adelphi, MD 20783

Commander
U.S. Army Test and Evaluation Command
ATTN: DRSTE-ME
DRSTE-TD
Aberdeen Proving Ground, MD 21005

Commander
U.S. Army White Sands Missile Range
ATTN: STEWS-AD-L
STEWS-ID
STEWS-TD-PM
White Sands Missile Range, NM 88002

Commander
U.S. Army Yuma Proving Ground
ATTN: Technical Library
Yuma, AZ 85364

Commander
U.S. Army Tropic Test Center
ATTN: STETC-TD, Drawer 942
Fort Clayton, Canal Zone

Commander
Aberdeen Proving Ground
ATTN: STEAM-MT-M, J. A. Feroli
STEAP-MT
STEAP-MT-G, R. L. Huddleston
Aberdeen, MD 21005

Commander
U.S. Army Cold Region Test Center
ATTN: STECR-OP-PM
APO Seattle 98733

Commander
U.S. Army Dugway Proving Ground
ATTN: STEDP-MT
Dugway, UT 84022

Commander
U.S. Army Electronic Proving Ground
ATTN: STEEP-MT
Fort Huachuca, AZ 35613

Commander
Jefferson Proving Ground
ATTN: STEJP-TD-I
Madison, IN 47250

Commander
U.S. Army Aircraft Development Test Activity
ATTN: STEBG-TD
Fort Rucker, AL 36362

President
U.S. Army Armor and Engineer Board
ATTN: ATZKOAE-TA
Fort Knox, KY 40121

President
U.S. Army Field Artillery Board
ATTN: ATZR-BDOP
Fort Sill, OK 73503

Commander
Anniston Army Depot
ATTN: SDSAN-QA
Anniston, AL 36202

Commander
Corpus Christi Army Depot
ATTN: SDSCC-MEE, Mr. Haggerty,
(Mail Stop 55)
Corpus Christi, TX 78419

Commander
Letterkenny Army Depot
ATTN: SDSLE-QA
Chambersburg, PA 17201

Commander
Lexington-Bluegrass Army Depot
ATTN: SDSLX-QA
Lexington, KY 40507

Commander
New Cumberland Army Depot
ATTN: SDSNC-QA
New Cumberland, PA 17070

Commander
U.S. Army Depot Activity
ATTN: SDSTE-PU-Q (2)
Pueblo, CO 81001

Commander
Red River Army Depot
ATTN: SDSRR-QA
Texarkana, TX 75501

Commander
Sacramento Army Depot
ATTN: SDSSA-QA
Sacramento, CA 95813

Commander
Savanna Army Depot Activity
ATTN: SDSSV-S
Savanna, IL 61074

Commander
Seneca Army Depot
ATTN: SDSSE-R
Romulus, NY 14541

Commander
Sharpe Army Depot
ATTN: SDSSH-QE
Lathrop, CA 95330

Commander
Sierra Army Depot
ATTN: SDSSI-DQA
Herlong, CA 96113

Commander
Tobyhanna Army Depot
ATTN: SDSTO-Q
Tobyhanna, PA 18466

Commander
Tooele Army Depot
ATTN: SDSTE-QA
Tooele, UT 84074

Director
DARCOM Ammunition Center
ATTN: SARAC-DE
Savanna, IL 61074

Naval Research Laboratory
ATTN: Code 5830, J. M. Krafft
Code 2620, Library
Washington, DC 20375

Air Force Materials Laboratory
Wright-Patterson Air Force Base
ATTN: AFML-LTM, W. Wheeler
AFML-LLP, R. Rowand
Wright-Patterson Air Force Base, OH 45433

DARCOM
ATTN: DRCQA, Mr. S. Lorber
DRCQA-E, Aida Estrella
Reba Gates
5001 Eisenhower Avenue
Alexandria, VA 22333

U.S. Army AMCCOM
ATTN: DRSMC-QAL, John Lipton
Rock Island, IL 61299

U.S. Army MICOM
ATTN: DRSMI-QET, R. Kevin Preston
Redstone Arsenal, AL 35898

David Jenkins
Intern Training Center
Red River Army Depot
Bldg 468
Texarkana, TX 75507

U.S. Army CECOM
ATTN: DRSEL-PA-M, George Panagous
DRSEL-PA-RT, Paul Kogut
Ft. Monmouth, NJ 07703

Jeffrey Cook
U.S. Army Plant Representative Office/
Hughes Aircraft Corporation
Mesa Field Office
5000 E. McDonell
Mesa, AZ 85201

U.S. Army ERADCOM
ATTN: DRDELEW-ES-PA, Russell Langan
Ft. Monmouth, NJ 07703

U.S. Army AVSCOM
ATTN: DRDAV-QP, Daniel Whyte
St. Louis, MO 85201

END

FILMED

9-81

DTIC